IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Title

Computer-Implemented System And Method For Lock Handling

Inventor

Charles S. Shorb

EV243779456US

TITLE

Computer-Implemented System And Method For Lock Handling

TECHNICAL FIELD

The present invention relates generally to computer-implemented accessing of resources and more particularly to lock handling in the accessing of resources.

BACKGROUND

In multi-threaded environments, threads compete against each other for resources. The resources which the threads wish to access may include a file, an I/O device, data stored in memory, etc. Current approaches to handling the accessing of resources tend to exhibit performance inefficiencies. For example, one approach includes using a mutex (mutual exclusion) algorithm for synchronizing multiple threads read and write access to shared resources. Once a mutex has been locked by a thread, other threads attempting to lock it will be blocked. When the locking thread unlocks (releases) the mutex, one of the blocked threads will acquire it and proceed. (See *Programming with POSIX Threads*, David R. Butenhof, Addison Wesley Longman, Inc., Copyright 1997, pages 241-253).

Another example involves use of a native operating system lock to protect the underlying lock data structure elements (See *Programming with POSIX Threads*, David R. Butenhof, pages 253-269). This approach uses a read and write wait queue to block waiting requests until it is safe to service the requests. Obtaining exclusive access to the internal data structure of the shared lock by first obtaining an exclusive lock is computationally expensive and thus performance suffers.

SUMMARY

In accordance with the disclosure provided herein, systems and methods are provided for handling access to one or more resources. Executable entities that are running substantially concurrently provide access requests to an operating system (OS). One or more traps of the OS are avoided through use of lock state information stored in a shared locking mechanism. The shared locking mechanism indicates the overall state of the locking process, such as the number of processes waiting to retrieve data from a resource and/or whether a writer process is waiting to access the resource.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram depicting software and computer components utilized in lock management;

FIGS. 2 and 3 are block diagrams depicting software and computer components utilized in lock management for a multi-threaded environment;

FIG. 4 is a flow chart depicting an operational locking scenario;

FIGS. 5 and 6 are bit data store diagrams depicting locking mechanisms;

FIG. 7 is a flow chart depicting a read lock acquisition example;

FIG. 8 is a flow chart depicting a read lock release example;

FIG. 9 is a flow chart depicting a write lock acquisition example;

FIG. 10 is a flow chart depicting a write lock release example;

FIGS. 11 and 12 are block diagrams depicting example applications utilizing a locking mechanism; and

FIG. 13 is a block diagram depicting a different lock system configuration.

DETAILED DESCRIPTION

FIG. 1 depicts at 30 a computer-implemented system for handling access to one or more resources 32. Multiple concurrently running executable entities 34 (e.g., processes, threads, etc.) compete against each other to access the resources 32. The executable entities 34 provide access requests 36 to a shared locking mechanism 40. Access 42 to the resources 32 is handled based upon information stored in a shared locking mechanism 40.

The shared locking mechanism 40 details where in the locking process the system 30 is. For example, the shared locking mechanism 40 may indicate the number of executable entities waiting to retrieve data from a resource 32. The lock state information stored by the mechanism 40 may also include whether an executable entity 34 wishes to update data associated with a resource 32. The mechanism's lock state information indicates when it is proper to grant access to a resource 32 and, in some situations, what type of access is to be performed.

The locking state information stored in the shared locking mechanism 40 allows the opportunity for one or more traps 38 of the computer's operating system to be avoided when a resource is to be accessed. Avoided traps include those interrupts provided by the operating system's kernel that involve mutex algorithms or other costly locking approaches when accessing requested resources. The capability to avoid relatively intensive operating system (OS) kernel traps increases performance. It should be understood that the shared locking mechanism 40 may decide to utilize the OS traps 38, and may use in certain situations a mutually exclusive locking technique provided by the operating system in order to provide access 42 to a resource 32. Still further, the system 30 may be used so as to avoid all or substantially all OS calls when accessing resources through the shared locking mechanism 40.

There are many types of environments within which an execution entity 34 may use a shared locking mechanism 40 to request and access a resource 32. As an example, these include multi-processing, multi-tasking, and multi-threaded environments. An execution entity 34 can include processes, threads, daemons, applets, servlets, ActiveX components, stand-alone applications, code that runs at the request of another program, etc.

FIG. 2 depicts a multi-threaded environment 50 where resources 32 require protection on a per thread basis. In this example, thread 1 (shown at 52) wishes to write to a resource 32, and thread 2 (shown at 54) wishes to read from a resource 32. Respectively from the threads (52, 54), a write request and a read request are routed to the shared locking mechanism 40. Due to knowledge of where in the locking process the system is, the shared locking mechanism 40 can minimize difficulties that may arise from the handling of the requests. As an illustration, the shared locking mechanism 40 may want, based upon the lock state information 56, to allow the read request and any others to succeed unless there is a write request active or pending. By examining the lock state information 56, the shared locking mechanism 40 can discern whether a resource write request is active or pending.

To assist a shared locking mechanism 40 in determining how to handle thread requests, the lock state information 56 may contain a wide variety of information. As an example, FIG. 3 shows that the lock state information 56 may include not only whether a write request is active but also information about any active or pending reader threads.

The shared locking mechanism 40 could use the lock state information 56 in combination with a rules mechanism 60. The rules mechanism 60 provides guidance as to how access to the resources 32 should proceed based upon the lock state information 56. Additionally, the rules mechanism 60 may be formulated to enhance performance. As an

illustration, obtaining exclusive access to the internal data structure of a shared lock by first obtaining an exclusive lock via OS traps 38 is typically computationally expensive. The rules mechanism 60 may offer an improvement in performance by providing the following locking guidelines when a request is being processed:

- Any number of read requests succeed in accessing a resource without using an exclusive lock unless there is a writer active or pending.
- Only one writer may be active at any given time.

It should be understood that many other different types of locking rules may be used, either as a replacement for the examples provided above or as an augmentation. For example, the rules may include the following: no preference is given when deciding which of the waiting read or write requests should be honored first. This rule allows fairness, such as eliminating starvation situations wherein read or write requests are processed with a higher priority with a result that they may in some situations not allow the other type of request to be timely processed (e.g., reader or writer starvation situations).

The lock state information 56 may be used to indicate when lock-related events are to be posted by one thread in order to notify another thread that its request can be serviced. For example, the lock state information 56 may contain status information as to whether a reader thread can post a lock release event to a writer thread. This prevents multiple reader threads from creating redundant posting of lock release events. It is noted that for a writer pending state a lock release event involves notification that a thread has completed its access of a resource. A lock ready event involves notification to a waiting writer thread that it can proceed.

FIG. 4 is a flow chart depicting an operational locking scenario. Start indication block 100 indicates that a request to access a resource is received at step 102. The request may

be a read or write request. In order to process the request, the lock state information is analyzed at step 104. The lock state information may be updated at this step to indicate a pending read or write request exists.

At step 106, the lock to the requested resource is acquired. The lock state information may be updated at this step to indicate that a read or write lock has been acquired. At step 108, the resource request is accessed based upon the analyzed lock state information. Step 110 releases the lock, and the lock state information may be updated to indicate that a read or write lock has been released. Processing for this iteration ends at end block 112.

FIG. 5 depicts at 200 an example of a lock status data store that could be used. The lock status data store 200 of FIG. 5 uses two classes of bits within the lock status: status bits 202 and bits 204 which form an integer subset containing the count of reader requests that have been granted and that need to be processed. In this example, the encapsulation of the locking state information as a single unit allows atomic hardware operations to quickly determine correct processing of the access request for the lock. It is noted that the term "atomic operation" is generally considered to be an operation which is allowed to start and finish without being interrupted.

A lock status data store 200 may use as many bits as desired to capture the locking state of the system. For example, FIG. 6 depicts a lock status data store 200 which uses three bits (220, 222, 224) as status bits 210 which are available for atomic access. The status bits 210 offer multi-threaded protection via a quick determination of a processing path to be used in place of an operating system (OS) mutex.

In the example shown in Fig. 6, the write pending/requested bit 220 is set only when a write request has been made. This provides an expedient method for determination of

whether the slower OS mutex lock is required. Because most shared lock's requests are typically for read requests, performance is improved whenever the more computationally involved OS mutex can be avoided. The lock status data store 200 allows an operating system to avoid the OS mutex approach (e.g., avoiding the OS mutex approach only when no writer is pending). Correspondingly, the lock status data store 200 can indicate that in some situations the OS mutex lock is to be used, such as when a writer request is pending or active.

In the lock status data store 200, the writer active bit 222 is set to indicate that a writer thread is currently active. This provides protection against reader threads posting an OS event after the writer thread has been activated. It is noted that an OS event is a signaling mechanism to notify one or more threads of an occurrence of a particular event (e.g., notification that a lock has been released).

The wait event posted bit 224 is set to indicate that a write request waits on a wait event to be posted before proceeding. The last active read operation will clear this bit prior to posting an event to ensure that one and only one post will occur; this posting will occur if the writer active bit is not set.

The remaining low order bits 230 are used for an active reader count in a normal 2's complement fashion. By tracking the reader count, the system knows whether readers are present and when the last reader exits. Any number of bits may be used for the reader count. In a 32 bit atomic integer, 29-bits are available for read requests, which allow for up to 536,870,912 read lock requests. It should be understood that the order of the bits provided in FIG. 6 serves as an example of a lock data structure and that the order of the bits may be altered in different ways while still achieving a safe, shared and mutually exclusive desired lock status data structure.

It is noted that many techniques may be used with the lock status data store 200, such as implementing shared access locks using basic OS primitives for those OS implementations not having native shared locks. For those that do, use of a lock status data store 200 can show increased performance over the native implementation by, among other things, using hardware atomic operations on a single status atomic integer in place of a slower OS mutex locking of a status data structure.

In this example, the shared lock may include the following elements encapsulated in a structure forming the basis of a shared lock: a native OS lock, a native OS event, and a host specific entity allowing for atomic integer operations which maintains the lock status. The host specific atomic entity may be an integer, or two integers where one serves as a barrier to modification of the other (to allow for machine level instruction spin-lock implementations).

The implementation may utilize host-level atomic routines for manipulation of the atomic entity, such as: an atomic get operation, which returns the current integer value of the atomic entity (on most hosts this may be the value integer portion of the atomic entity); an atomic set operation, which attempts to perform a compare/exchange operation on the atomic, and allows for indication of success or failure of the operation; an atomic add operation, which adds a value to the current atomic entity value and returns the new value; and an atomic sub operation which is typically an atomic add operation with the addend being the negative of the input to the atomic sub operation.

FIG. 7 is a flow chart depicting a read lock acquisition example 300. Due to the fact that shared locks are primarily used in read lock mode, the implementation in this example is designed to favor read requests as shown in step 302 with an initial increment of the lock status.

If the lock status value shows that there is no writer request pending (or active) as determined at decision step 304, then the read request is said to have succeeded as shown at indicator 314.

If there is a pending or active write request as determined by decision step 304, the read request unregisters its lock status that was made in step 302. In other words, the process backs out of the increment. This is done at step 306 by following the procedure described in reference to FIG. 8 below. Following the read lock release, the read request waits on the lock OS mutex in step 308. After acquiring the lock OS mutex, the lock status can safely be incremented in step 310. The lock OS mutex is then released at step 312, and the read lock request is now satisfied as shown at indicator 314.

FIG. 8 is a flow chart depicting a read lock release example 400. First the lock status is decremented at step 402. An attempt is made at step 404 to set the event post bit of the lock status. This may be done using an atomic set operation with an expected value such that the value will be set only when a write request is pending and not active, and the read request count is zero. If this value is indeed set at step 404 as determined by decision block 406, then an OS event is posted at step 408 to ensure that a waiting writer is allowed to run. After posting the OS event at step 408, or if the atomic set operation failed at step 404, then the read request is deemed successful as shown at indicator 410. It is noted that setting the lock event posted bit at step 404 may have failed because another read request had previously posted the event. Setting of the lock event posted bit ensures protection against multiple posts of the OS event and against the writer being active.

FIG. 9 is a flow chart depicting a write lock acquisition example 500. First, the writer attempts to acquire a lock OS mutex at step 502. This provides that there is only one active writer at any given time and thus protects a writer from other writers. An attempt is made

to set the lock status for both writer pending, writer active and not-posted bits at step 504. If the lock status was set as determined by decision step 506, then the write lock has been acquired as shown by indicator 520.

However if the lock status was not set as determined by decision step 506, then this indicates that the read count is non-zero (i.e., readers are present), and the lock OS wait event is cleared at step 508 in preparation for a possible wait for an event that indicates that the writer can proceed. The OS event may be associated with the threads themselves instead of the lock data structure. However, there may be certain situations wherein an OS event may be associated with the lock status. It should be understood that based upon the situation at hand an OS event may be associated with the lock data structure.

The write pending (i.e., write requested) and not-posted bits are set at step 510, such as by using an atomic add operation. The lock status reader count is again checked at step 512, and if the reader count is zero, the write request is safe to honor, and step 518 marks the writer active bit of the lock status. After step 518 finishes, the write lock is deemed acquired as shown at indicator 520.

If decision step 512 determines that the lock status read count is not equal to zero, then read requests are still outstanding, and the write request waits at stop 514 for a lock OS event. It is noted that waiting on the lock OS event at step 514 (on FIG. 9) may end when a read request posts the OS event at step 408 (on FIG. 8).

After step 514, the writer active and not-posted bits are set in the lock status at step 516. The not-posted bit is set as an indicator that the lock OS event should not be waited upon when the write lock release occurs. After step 516 finishes, the write lock is deemed acquired as shown at indicator 520.

FIG. 10 is a flow chart depicting a write lock release example 600. At step 602, a check of the lock status for the not-posted flag is performed. If the not-posted flag is set, then the lock OS event is waited for at step 604. This ensures that the read request that has knowledge of the OS event is given the chance to post it. This also protects against a read request (knowing of the OS event to post) being preempted prior to posting, and returning to post an OS event that is setup for a different future pending write request. The remaining active bits are cleared at either steps 606 or 608 depending upon the branching that occurred at decision step 602. Step 606 involves the clearing of two bits (i.e., the writer and active bits) in the lock status, whereas step 608 involves the clearing of three bits (i.e., the writer, active and not-posted bits). This ensures that all of the bits in the write requested and writer active positions are cleared before the OS mutex is released. After the bits have been cleared, the lock OS mutex can be safely released at step 610, and the write lock is released as shown at indicator 612.

It is noted that the flow charts discussed herein may have more or less steps than what are disclosed herein in order to suit the situation at hand. For example in FIGS. 7 and 10, the processing may have a slight read lock acquisition bias, in that after the lock status has its write pending bit cleared (e.g., in steps 606 or 608) a new read lock acquisition request at step 304 can proceed ahead of all pending requests (e.g., steps 308, 502). This should be acceptable in most situations as most shared lock implementations already have a read bias, whereas later reads can progress ahead of pending write requests. However, it should be understood that the processing may also, if the situation arises, be modified to have a write lock acquisition bias. It should be further understood that even if a bias is used (whether it is a reader or writer bias), fairness is achieved -- e.g., a starvation situation or a deadlock situation does not occur due to use of the lock status data structure.

FIGS. 11 and 12 are block diagrams depicting some of the wide ranges of resources to which a locking system may be directed. FIG. 11 illustrates a shared locking mechanism 40 being used to coordinate access to a word processing document 700. In this example, multiple users wish to read the word processing document 700 while one or more users wish to update the document 700. The shared locking mechanism 40 guides based upon the teachings disclosed herein how access to the word processing document 700 occurs for each of these requests.

While a single resource is shown in FIG. 11, it should be understood that a shared locking mechanism 40 may be used with multiple resources, such as shown in FIG. 12. FIG. 12 illustrates the shared locking mechanism 40 being used to coordinate access to multiple input/output (I/O) devices 720. Still further, many other types of resources can be managed by the shared lacking mechanism 40, such as memory locations, queues, etc.

While examples have been used to disclose the invention, including the best mode, and also to enable any person skilled in the art to make and use the invention, the patentable scope of the invention is defined by the claims, and may include other examples that occur to those skilled in the art. As an illustration, although the flow charts of FIGS. 7-10 discuss the explicit lock case for both read and write requests, the processing may be expanded to include other cases, such as the 'try' lock case. This may include, but is not limited to, returning a LOCK_BUSY code when the lock cannot be immediately obtained.

As yet another example of the many applications and extensions of the system, FIG. 13 depicts that a software module 800 may act as an interface between the shared locking mechanism 40 and the software mechanisms 40 and 60, as well as between the shared locking

mechanism 40 and OS trap(s) 38. This may help to hide implementation details involving the shared locking mechanism 40, rules mechanism 60, and the OS trap(s) 38.

It is noted that processes and/or threads that wish to access the resources may enter any number of states while waiting to access a resource. They may enter a hibernation/sleep state or if allowed, may continue executing until the resource becomes available. Still further, they may execute a spin loop and continue to request access to the resource until it is released.

It is further noted that the systems and methods disclosed herein may be implemented on various types of computer architectures, such as for example on a single general purpose computer or workstation, or on a network (e.g., local area network, wide area network, or internet), or in a client-server configuration, or in an application service provider configuration. As an illustration, the requesting processes may all reside locally on the same computer as the shared locking mechanism, or may be distributed on remote computers, or may reside both on a local computer as well as on one or more remote computers. As another example of the wide scope, different OS atomic operations may be utilized with the systems and methods, and the OS atomic operations may dictate the implementation of the lock status data structure (e.g., for some hosts the atomic nature of tracking requests in the lock status data structure may be implemented by a load/clear approach; in a load/clear architecture, the tracking of read/write status may be handled by a separate locking integer protecting the actual status information. Note: This architecture would not require 'backing out changes' as was shown in FIG. 8.)

The systems' and methods' data may be stored as one or more data structures in computer memory depending upon the application at hand. The systems and methods may be

provided on many different types of computer readable media including instructions being executable by a computer to perform the system and method operations described herein.

The computer components, software modules, functions and data structures described herein may be connected directly or indirectly to each other in order to allow the flow of data needed for their operations. It is also noted that a software module may include but is not limited to being implemented as one or more sub-modules which may be located on the same or different computer. A module may be a unit of code that performs a software operation, and can be implemented for example as a subroutine unit of code, or as a software function unit of code, or as an object (as in an object-oriented paradigm), or as an applet, or as another type of computer code.

It should be understood that as used in the description herein and throughout the claims that follow, the meaning of "a," "an," and "the" includes plural reference unless the context clearly dictates otherwise. Finally, as used in the description herein and throughout the claims that follow, the meanings of "and" and "or" include both the conjunctive and disjunctive and may be used interchangeably unless the context clearly dictates otherwise; the phrase "exclusive or" may be used to indicate situation where only the disjunctive meaning may apply.